Servant Locator Example

To illustrate the servant locator concepts outlined so far, let us examine a (very simple) implementation. Consider that we want to create an electronic phone book for the entire world's telephone system (which, clearly, involves a very large number of entries, certainly too many to hold the entire phone book in memory). The actual phone book entries are kept in a large database. Also assume that we have a search operation that returns the details of a phone book entry. The Slice definitions for this application might look something like the following:

Slice

```
struct Details {
    // Lots of details about the entry here...
};

interface PhoneEntry {
    idempotent Details getDetails();
    idempotent void updateDetails(Details d);
    // ...
};

struct SearchCriteria {
    // Fields to permit searching...
};

interface PhoneBook {
    idempotent PhoneEntry* search(SearchCriteria c);
    // ...
};
```

The details of the application do not really matter here; the important point to note is that each phone book entry is represented as an interface for which we need to create a servant eventually, but we cannot afford to keep servants for all entries permanently in memory.

Each entry in the phone database has a unique identifier. This identifier might be an internal database identifier, or a combination of field values, depending on exactly how the database is constructed. The important point is that we can use this database identifier to link the proxy for an Ice object to its persistent state: we simply use the database identifier as the object identity. This means that each proxy contains the primary access key that is required to locate the persistent state of each Ice object and, therefore, instantiate a servant for that Ice object.

What follows is an outline implementation in C++. The class definition of our servant locator looks as follows:

C++

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& c, Ice::LocalObjectPtr& cookie);

    virtual void finished(const Ice::Current& c, const Ice::ObjectPtr& servant,
                          const Ice::LocalObjectPtr& cookie);

    virtual void deactivate(const std::string& category);
};
```

Note that `MyServantLocator` inherits from `Ice::ServantLocator` and implements the pure virtual functions that are generated by the `slice2cpp` compiler for the `Ice::ServantLocator` interface. Of course, as always, you can add additional member functions, such as a constructor and destructor, and you can add private data members as necessary to support your implementation.

In C++, you can implement the `locate` member function along the following lines:

C++

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& c, Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error&)
        return 0;
    }

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}
```

For the time being, the implementations of `finished` and `deactivate` are empty and do nothing.

The `DB_lookup` call in the preceding example is assumed to access the database. If the lookup fails (presumably, because no matching record could be found), `DB_lookup` throws a `DB_error` exception. The code catches that exception and returns zero instead; this raises `ObjectNotExistException` in the client to indicate that the client used a proxy to a no-longer existent Ice object.

Note that `locate` instantiates the servant on the heap and returns it to the Ice run time. This raises the question of when the servant will be destroyed. The answer is that the Ice run time holds onto the servant for as long as necessary, that is, long enough to invoke the operation on the returned servant and to call `finished` once the operation has completed. Thereafter, the servant is no longer needed and the Ice run time destroys the smart pointer that was returned by `locate`. In turn, because no other smart pointers exist for the same servant, this causes the destructor of the `PhoneEntryI` instance to be called, and the servant to be destroyed.

The upshot of this design is that, for every incoming request, we instantiate a servant and allow the Ice run time to destroy the servant once the request is complete. Depending on your application, this may be exactly what is needed, or it may be prohibitively expensive — we will explore designs that avoid creation and destruction of a servant for every request shortly.

In Java, the implementation of our servant locator looks very similar:

Java

```
public class MyServantLocator implements Ice.ServantLocator {

    public Ice.Object locate(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
        // Get the object identity. (We use the name member
        // as the database key.
        String name = c.id.name;

        // Use the identity to retrieve the state
        // from the database.
        //
        ServantDetails d;
        try {
            d = DB.lookup(name);
        } catch (DB.error e) {
            return null;
        }

        // We have the state, instantiate a servant and return it.
        //
        return new PhoneEntryI(d);
    }

    public void finished(Ice.Current c, Ice.Object servant, java.lang.Object cookie)
    {
    }

    public void deactivate(String category)
    {
    }
}
```

The C# implementation is virtually identical to the Java implementation, so we do not show it here.

All implementations of `locate` follow the pattern illustrated by the previous pseudo-code:

1. Use the `id` member of the passed `Current` object to obtain the object identity. Typically, only the `name` member of the identity is used to retrieve servant state. The `category` member is normally used to select a servant locator. (We will explore use of the category member shortly.)
2. Retrieve the state of the Ice object from secondary storage (or the network) using the object identity as a key.
    - If the lookup succeeds, you have retrieved the state of the Ice object.
    - If the lookup fails, return null. In that case, the Ice object for the client's request truly does not exist, presumably, because that Ice object was deleted earlier, but the client still has a proxy to the now-deleted object.
3. Instantiate a servant and use the state retrieved from the database to initialize the servant. (In this example, we pass the retrieved state to the servant constructor.)
4. Return the servant.

Of course, before we can use our servant locator, we must inform the adapter of its existence prior to activating the adapter, for example (in Java or C#):

```
MyServantLocator sl = new MyServantLocator();
adapter.addServantLocator(sl, "");
```

Note that, in this example, we have installed the servant locator for the empty category. This means that `locate` on our servant locator will be called for invocations to any of our Ice objects (because the empty category acts as the default). In effect, with this design, we are not using the `category` member of the object identity. This is fine, as long as all our servants all have the same, single interface. However, if we need to support several different interfaces in the same server, this simple strategy is no longer sufficient.

See Also

- Servant Locators